Table of Contents

LESSON: The Ant Build Tool 1

Subjects 2

History − Managing the Build 3

Ant Principles 4

First Ant − Step 1: Java sources 5

First Ant − Step 2: Ant build file 6

First Ant − Step 3: Execute Ant 7

Roundup 8

RESOURCES 9

LESSON: The Ant Build Tool

author: Just van den Broecke − just@justobjects.nl
organization: Just Objects − www.justobjects.nl

Slides

1. Subjects
2. History − Managing the Build
3. Ant Principles
4. First Ant − Step 1: Java sources
5. First Ant − Step 2: Ant build file
6. First Ant − Step 3: Execute Ant
7. Roundup

Labs

Resources

Subjects

• History • Quickstart: First Ant build • Installation and execution • Buildfile anatomy: project, targets, tasks, properties • Built−in tasks • Optional tasks • Advanced: User−defined tasks	
The Ant Build Tool	1

(From the Ant user manual)

Ant is a Java−based build tool. In theory, it is kind of like make, without make's wrinkles.

Why?

Why another build tool when there is already make, gnumake, nmake, jam, and others? Because all those tools have limitations that Ant's original author couldn't live with when developing software across multiple platforms. Make−like tools are inherently shell−based: they evaluate a set of dependencies, then execute commands not unlike what you would issue on a shell. This means that you can easily extend these tools by using or writing any program for the OS that you are working on; however, this also means that you limit yourself to the OS, or at least the OS type, such as Unix, that you are working on.

Makefiles are inherently evil as well. Anybody who has worked on them for any time has run into the dreaded tab problem. "Is my command not executing because I have a space in front of my tab?!!" said the original author of Ant way too many times. Tools like Jam took care of this to a great degree, but still have yet another format to use and remember.

Ant is different. Instead of a model where it is extended with shell−based commands, Ant is extended using Java classes. Instead of writing shell commands, the configuration files are XML−based, calling out a target tree where various tasks get executed. Each task is run by an object that implements a particular Task interface.

Granted, this removes some of the expressive power that is inherent in being able to construct a shell command such as `find . −name foo −exec rm {}`, but it gives you the ability to be cross−platform − to work anywhere and everywhere. And hey, if you really need to execute a shell command, Ant has an <exec> task that allows different commands to be executed based on the OS it is executing on.

History − Managing the Build

- Primary tool: "make"
  - ♦ Complex, in particular for different OSs/platforms
  - ♦ Error−prone, e.g. spaces
- Java introduced cross platform software
  - ♦ make not suitable anymore (platform dependent)
  - ♦ open source development required platform−independent builds
- Enter Ant
  - ♦ started as build tool for Sun's servlet/JSP development
  - ♦ Sun's servlet/JSP development carried over to Jakarta Tomcat project
  - ♦ through Jakarta projects now the defacto build tool in the Java world

(From the A beginners guide to Ant by Steve Loughran)

Once upon a time, way back when you had to rewrite applications to run on different operating systems, and recompile applications to run on different processors, the whole tasks of "managing the build" was such a complex operation that large projects had people working on the task full time. Their primary tool was 'make', a powerful tool which could mix inference rules (what commands to issue to produce a .o object file from a .c C source file), with explicit dependencies.

A cross platform makefile is even more complex, as the commands to perform various tasks vary from system to system. That is why imake came in to being −a program which generates the appropriate makefile for a platform by actually trying out the software installed on the system to see what works and what does, then generating a platform specific makefile. Cross platform development was never fun, especially when testing came in to the equation.

Java changed the whole notion of cross platform software. Now you could run a program on many different operating systems and microprocessors. But for a long time the actual development process lagged. You could stick with make −and require everyone to develop on the same platform, or use an IDE −and require everyone to use the same IDE. These were viable options in small team projects, but not in the open source world.

Ant changed the whole notion of cross platform software development. The funny thing is −this was almost (but not entirely) by accident.

Ant Principles

- Conceptually Ant mimics make
- Build files specified in XML (Makefile equivalent)
  - ♦ hierarchical structure ideal for target/task dependency graph
  - ♦ extensible format
- General build command ("ant") written in Java (make equivalent)
  - ♦ interprets build file
  - ♦ executes build tasks (see next)
- Wide range of build tasks (Swiss Army Knife)
  - ♦ basically any task from checking out sources up to testing and deployment
  - ♦ core: compilation, javadoc, file management, jar/tar/zip, ...
  - ♦ optional: FTP, JUnit, rpm, ...
  - ♦ user−defined: plugin−like tasks developed by you using Ant Java API

The Ant Build Tool	3

Ant is similar to and at the same time different from make. While the make model is extended with shell−based commands (e.g. cc), Ant is extended using Java classes. Instead of writing shell commands, the configuration files are XML−based, calling out a target tree where various tasks get executed. Each task is run by an object that implements a particular Task interface.

Granted, this removes some of the expressive power that is inherent in being able to construct a shell command such as `find . −name foo −exec rm {}`, but it gives you the ability to be cross−platform − to work anywhere and everywhere. And if you really need to execute a shell command, Ant has an <exec> task that allows different commands to be executed based on the OS it is executing on.

First Ant − Step 1: Java sources

This represents your application sources

```
 1: // A basic "Java Application".
 2: public class FirstApp {
 3:
 4:     public static void main (String [] args) {
 5:         FirstClass firstClass = new FirstClass();
 6:         firstClass.doSomething();
 7:     }
 8: }
```

```
 1: // A basic Java class.
 2: public class FirstClass {
 3:
 4:     public FirstClass() {
 5:     }
 6:
 7:     public void doSomething() {
 8:             System.out.println("Hello Ant");
 9:     }
10: }
```

First Ant − Step 2: Ant build file

```
 1: <?xml version="1.0"?>
 2:
 3: <project name="first" default="jar" basedir=".">
 4:
 5:   <property name="class.dir"  value="${basedir}/class" />
 6:   <property name="dist.dir"  value="${basedir}/dist" />
 7:
 8:   <target name="compile" >
 9:     <mkdir dir="${class.dir}" />
10:     <javac srcdir="${basedir}" destdir="${class.dir}" />
11:   </target>
12:
13:   <target name="jar" depends="compile">
14:     <mkdir dir="${dist.dir}" />
15:     <jar jarfile="${dist.dir}/first.jar" basedir="${class.dir}" />
16:   </target>
17:
18:   <target name="clean" >
19:     <delete dir="${class.dir}" />
20:     <delete dir="${dist.dir}" />
21:   </target>
22:
23: </project>
```

project	the top−level element
target	a unit to be built through the contained tasks
tasks	specific units of work to perform (javac, jar, etc)
property	(optional) properties (name/value) global to entire project file.

Notice an Ant build file doesn't use a DTD. The reason is to provide extensibility for optional/user−defined tasks.

First Ant − Step 3: Execute Ant

Assuming Ant is installed

- Use the ant.bat/sh script or directly
  - ♦ script: ant.sh [options] [target]
  - ♦ directly: java −Dant.home=c:\ant org.apache.tools.ant.Main [options] [target]
- When executed without options or target
  - ♦ searches for build.xml in current dir
  - ♦ attempts building default target (jar in example)
- Example targets
  - ♦ ant compile runs target compile
  - ♦ ant clean runs target clean
- Example options
  - ♦ ant −buildfile project.xml use build file project.xml
  - ♦ ant −Dhome.dir=/home/just/holycow use given value for property home.dir

Roundup

- We have seen the very basics of Ant
  - Buildfile anatomy: project, targets, tasks, properties
  - Executing ant: arguments for options and targets
- Next steps
  - getting to know the specific core and optional tasks
  - dealing with files, directories, paths and filters
  - constructing the build file(s) for your project
  - extending Ant with tasks implemented by you
- Tips
  - the Ant user manual gives a valuable reference
  - study build files from other projects like those on sourceforge.net and jakarta.apache.org

RESOURCES

[jakarta−1] The Apache Jakarta Project ; *Apache Ant Project Page* ; Apache Ant is a Java−based build tool. In theory, it is kind of like Make, but without Makes wrinkles.
http://jakarta.apache.org/ant

[ant−1] Norman Chaffee ; *Building with Ant (three−part article)* ; See how to avoid many of the common growing pains of application development by using simple tools, like Ant and JUnit, and a simple directory structure. CowNote: great introduction with lots of detail on developing a build file.
http://softwaredev.earthweb.com/java/sdjojta/article/0,,12401_989631,00.html

[ant−2] Steve Loughran ; *A beginners guide to Ant* ; A quick introductory tutorial to the Ant build tool. It is intended for people starting out with Ant and Java development, and aims to provide enough detail to get started.
http://www.iseran.com/Java/ant/tutorial/ant_tutorial.html

[ant−3] Larry Port ; *Larrys Ant Tutorial* ; Ant is a utility using xml that was developed in response to frustration with makefiles. In a nutshell, you write an xml document, "build.xml" which builds your programs and checks for what depends on what. This is a get−up−and−running page, so for specific answeres and more detailed info please visit
http://rlab.cs.nyu.edu/~lport/antTutorial.html